

# How to rock the house with C++ in 2011

by Frédéric Dubouchet and Philippe Vaucher

# Topics

- Audience poll: do you know C++ already? other languages?
- General C++ intro
- General boost intro
- C++11 overview

# General intro on C++

- History
- Feature set
- Myths debunking
- Performance
- User base and success story

# C++ History

- Invented by Bjarne Stroustrup in 1979
  - First compiler 1983 (simply generated C code)
  - Specifications 1998 - 2003 - 2011
- Extended by a committee (ISO/IEC\_JTC1/SC22/WG21)
- Meant as a better C: classes, virtual functions, operator overloading, multiple inheritance, templates, exceptions, etc
- Modern language without performances compromises ("Zero Overhead Policy" -> You don't pay for what you don't use (e.g exceptions)).
- Multi-paradigm language: procedural, OO, functional...
- Because it's 30 years old and it's very permissive a lot of crap & good c++ styles codebase exist.

# Featureset

- Compatibility with C (access to libraries)
- Strong typing / function overloading
- Operators overloading
- Templates
- Standard Template Library (algorithms & data structures)

# Featureset - Strong Typing/Overloading

Allows to dispatch to functions with the same name based on the type.

```
int add(int a, int b)
{
    return a + b;
}
```

```
string add(string a, string b)
{
    return a.concat(b);
}
```

```
int main()
{
    int eight = add(3, 5);           // calls add(int, int)
    string helloworld = add("hello", "world"); // calls add(string, string)
}
```

# Featureset - Operator overloading (Part I)

When you write **a + b**, the compiler does **a.operator+(b)**

If the member function doesn't exist, it calls **operator+(a, b)**

```
string operator+(string a, string b)
{
    return a.concat(b);
}
```

```
string x, y;
string z = x + y;
```

By overloading operators, you can give intuitive behavior to your objects. You can overload a lot of operators:

- arithmetics (+, -, \*, etc)
- comparison (>, ==, <, >=, etc)
- logical (!, &&, ||, etc)
- bitwise (&, |, ^, ~, etc)
- conversion (int(), float(), etc)
- and more! (=, <<, ->, [], (), \*, &, ++, --, new, delete, etc)

# Featureset - Operator overloading (Part II)

You can make objects that behaves as functions (function-objects)

```
struct minmax
{
    sum() : min(INT_MAX), max(0) {}
    void operator()(int n)
    {
        if(n > max) max = n;
        if(n < min) min = n;
    }
    int min;
    int max;
};
```

```
minmax finder;
int arr[] = {1,2,3};
for(int i = 0; i != 3; ++i)
    finder(arr[i]);
int min = finder.min;
int max = finder.max;
```

This allows for function with states that don't use globals or statics.

# Featureset - Templates (part I)

It helps to see it as a macro generating code for you when you instantiate it.

```
template <class T>
T add(T a, T b)
{
    return a + b;
}

int main()
{
    int eight = add(3, 5);           // generate add<int>
    double three_dot_three = add(1.1, 2.2); // generates add<double>
    string helloworld = add("hello", "world"); // generates add<string>
}
```

Templates are duck typing!

You can also template a whole class and provide generic containers that way (array<int> ai; array<string> as;)

# Featureset - Templates (part II)

You can specialize them for a certain type/value. This allows for metaprogramming at compile time in a functional way.

```
template <unsigned N>
struct factorial
{
    enum { value = N * factorial<N-1>::value };
};

template <>
struct factorial<0>
{
    enum { value = 1 };
};

int main()
{
    int twenty_four = factorial<4>::value; // compile-time!!!
}
```

It allows to specialize containers/algorithms for a certain type and use the generic form for others.

# Featureset - STL

Basic structure:

- containers: templates classes implementing arrays, linked lists, queues, associative containers, etc
- each container provide iterators & share a common interface (`push_back()`, `size()`, etc).
- algorithms only take template iterators parameters.

Implications:

- if you write a new algorithm that takes iterators, it works for all containers.
- if you write a new container offering iterators, all algorithms works with it.

# Myths debunking

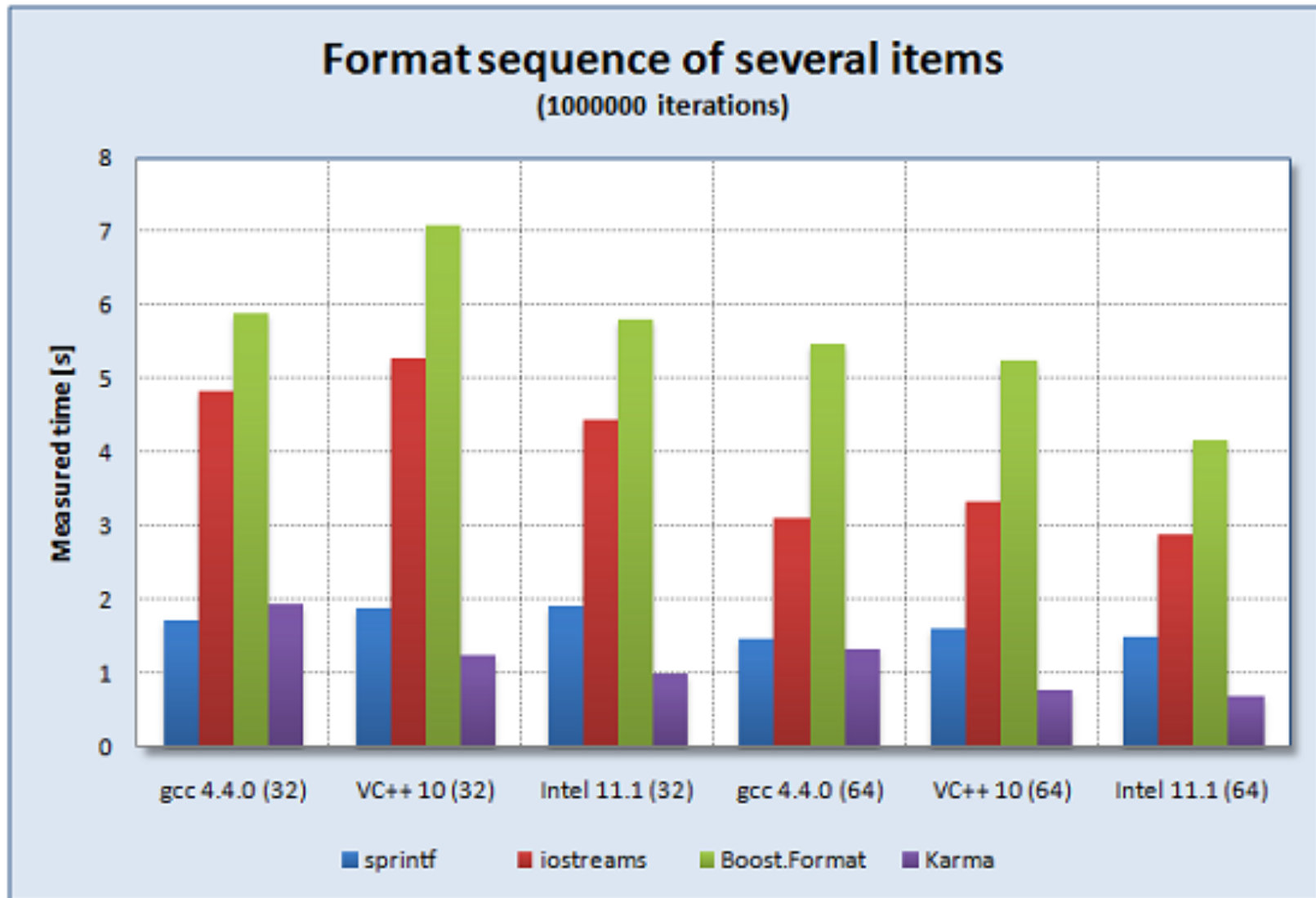
- GC is better (when RAI is ++better)
- Operator overloading implies you never know what the code will do (example "a + b" can send an email).
- Difficult to write / non-portable
- Not used anymore / soon (existing codebase is huge!)

*"C++ is my favorite garbage collected language because it generates so little garbage"*

—Bjarne Stroustrup

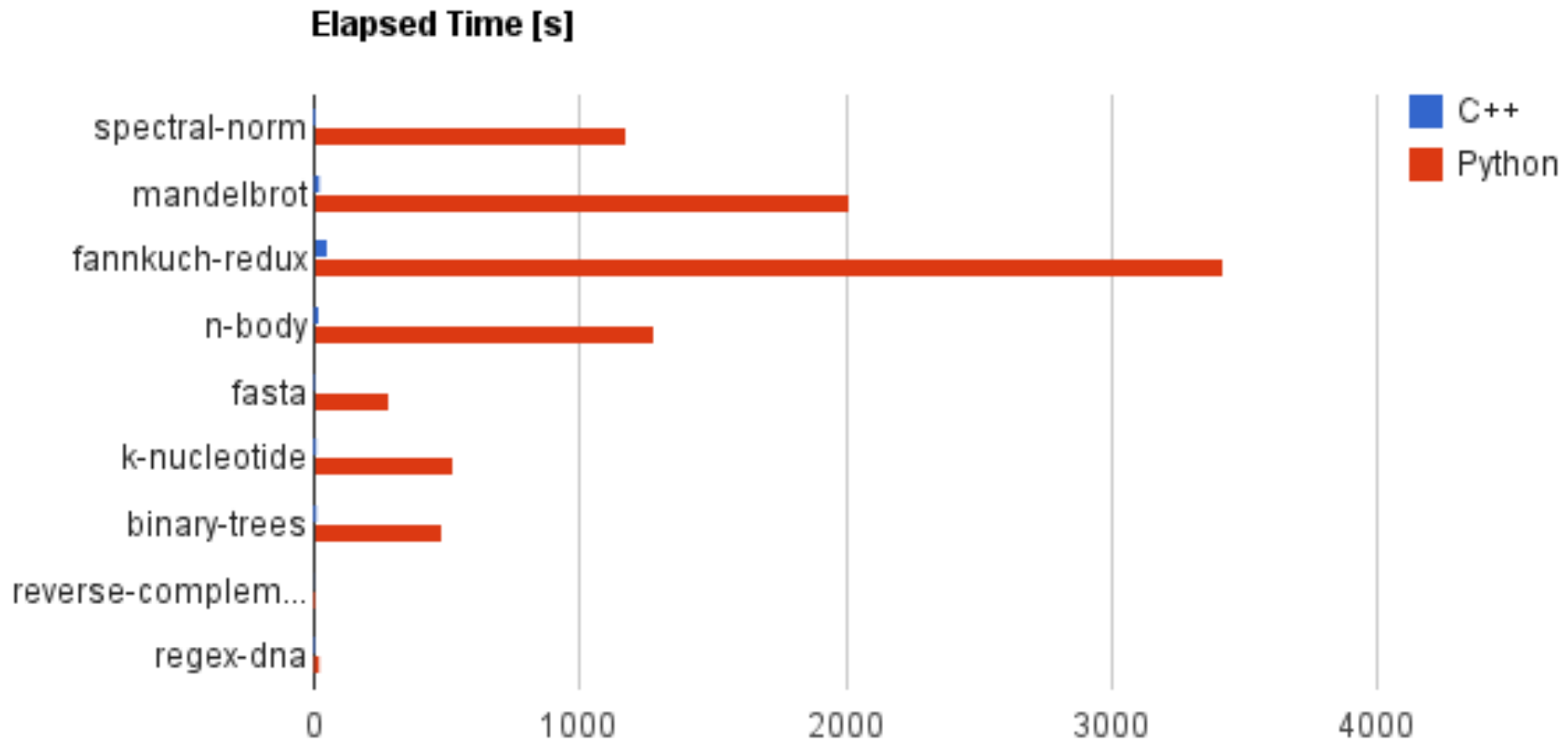
# Performance (part I)

boost::karma faster than sprintf() !



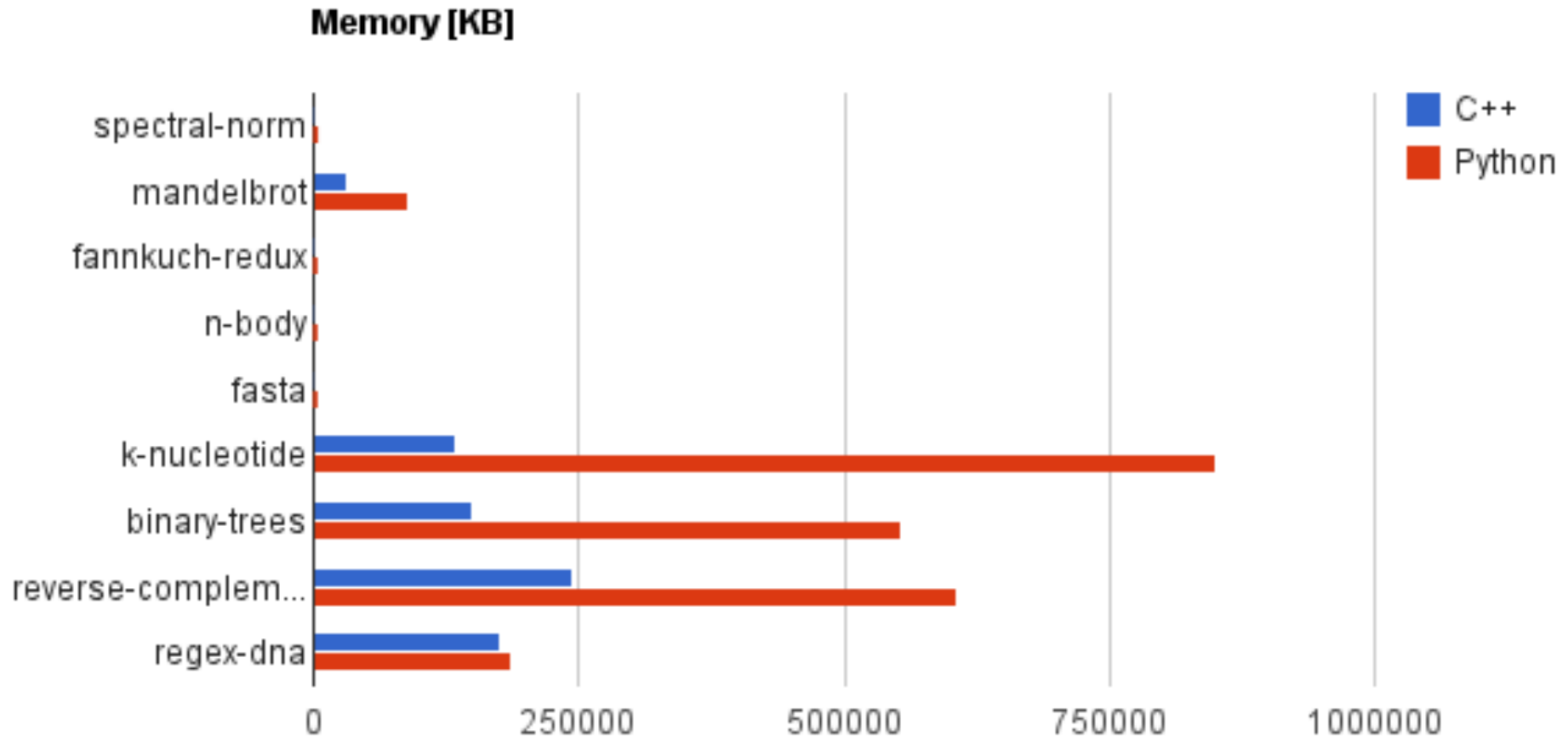
# Performance (part II)

c++ is much faster than interpreted languages (10x-100x)



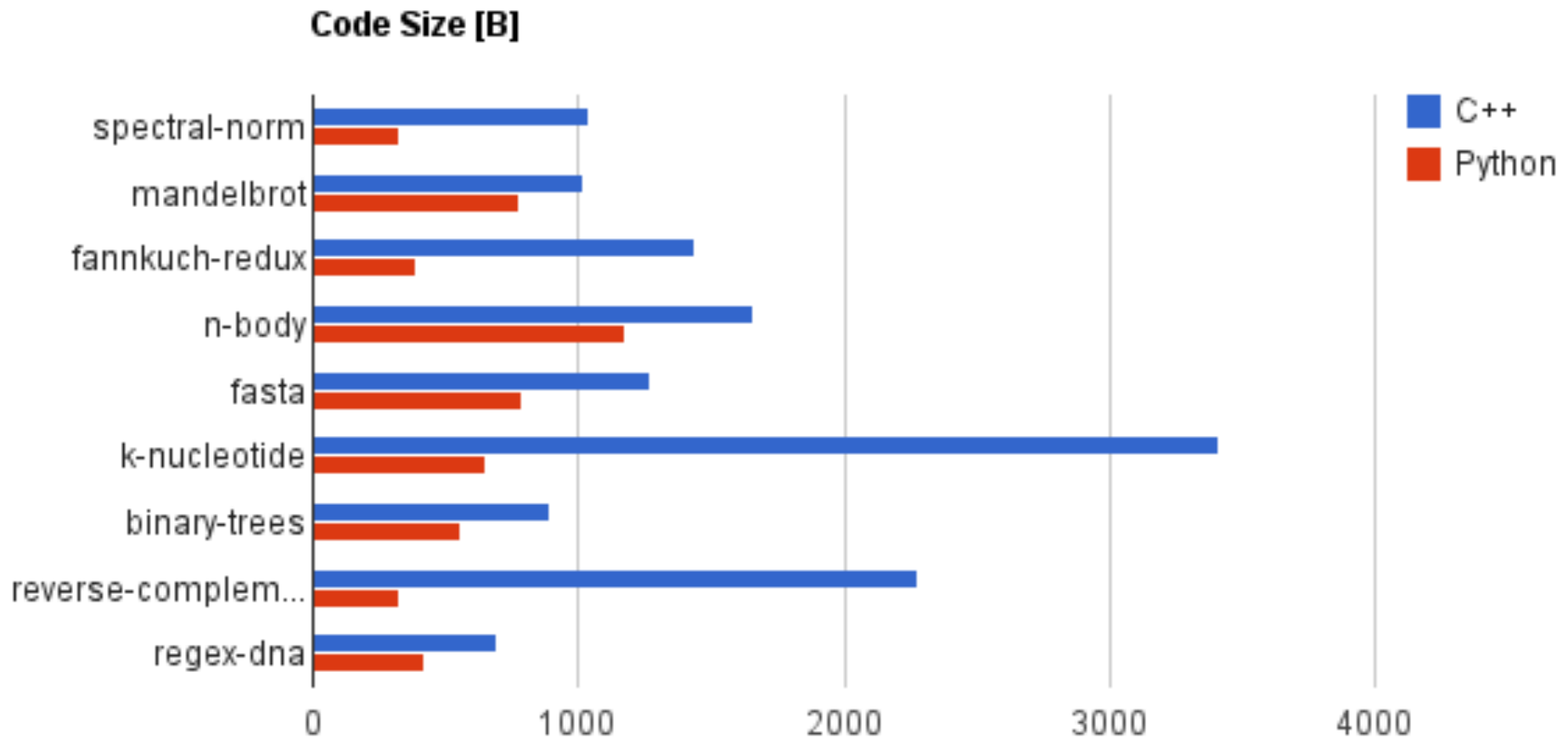
# Performance (part III)

c++ consumes less memory than interpreted languages (2x-10x)



# Performance (part III)

c++ needs more code than interpreted languages (1x-7x)



# User base and success stories

- Second most cited language (after C) on the Internet
- 3rd most used language (after Java & C)
- C++ runs the world!
  - All Browser (Mozilla, Safari, Chrome, IE, Opera,...)
  - All Big Games (Blizzard, ID, Valve, Bioware,...)
  - Most Big Internet companies (FB, Google,...)

*"There are only two kinds of languages: the ones people complain about and the ones nobody uses."*

—Bjarne Stroustrup

# General Boost intro

- What is boost?
- some examples
  - `boost::shared_ptr`
  - `boost::function`
  - `boost::bind`
  - `boost::asio`
  - `boost::serialize`
- other libraries

# What is boost

- Boost is a set of **portable** C++ libraries helping with text processing, containers, iterators, algorithms, function objects and higher-order programming, generic programming, template metaprogramming, preprocessor metaprogramming, concurrent programming, math and numerics, correctness and testing, data structures, image processing, input/output & inter-language support
- extends the STL
- Lots of boost will be in C++11 (see TR1, TR2)
- Boost used as a unit test for compilers (clang proudly announces it's now mature enough to build boost)

# boost::shared\_ptr

Generic ref-counted smart pointer

## Problem:

```
int main()
{
    FILE* f = 0;
    try
    {
        f = fopen("foo.txt", "r");
        process(f);
        fclose(f);
    }
    catch(const std::exception& e)
    {
        fclose(f);
    }
}
```

## Solution:

```
int main()
{
    shared_ptr<FILE> f(fopen("foo.txt", "r"), &fclose);
    process(f);
}
```

The custom deleter allows you to work with pretty much any resource.

# boost::function

Generic callback functor that can link to free functions  
OR function objects.

It greatly helps to write systems with callbacks as you  
don't need to duplicate code essentially doing the same  
thing.

```
int add(int, int);  
struct funky_adder;  
int callme(boost::function<int(int, int)> func)  
{  
    return func(1, 2);  
}
```

```
callme(&add);  
callme(funky_adder());
```

# boost::bind

Generic function-object generator (function adapters)

```
int add(int a, int b);
```

```
int main()
```

```
{
```

```
    boost::function<int(int)> give_me_5 = boost::bind(&add, 2, 3);
```

```
    int five = gimme5();
```

```
    boost::function<int(int)> add3 = boost::bind(&add, _1, 3);
```

```
    int seven = add3(4);
```

```
}
```

`_1`, `_2`, etc are **placeholders**, they represent arguments and use **expression templates**.

Bind allows you to store a `vector<function<void()>>` and just `bind()` everything into it for later use.

# boost::asio

- pro-actor based library active co-functions
- best possible performances for network.
- compatible with stream interface
- Interface for stdio, signal, UDP, TCP, ICMP, SSL
- use the best possible multiplexing facilities on specific platform
  - select (as fall back)
  - poll/epoll
  - kqueue
  - overlapped I/O
- facilities for memory management and boost::thread

# boost::serialize

Serialize any possible structures and class over into

- XML
- Binary
- plain text

## Example:

```
foobar m;  
std::stringstream ss("");  
boost::archive::binary_oarchive boa(ss);  
boa << m;
```

# More...

- boost::program\_options
- boost::spirit
- boost::fusion
- boost::python
- boost::filesystem
- boost::thread
- boost::iterators
- boost::signals
- boost::test
- boost::statechart
- boost::any
- boost::MPL
- boost::regex

- boost::smart\_ptr
- boost::variant
- and many many more!

# C++11 overview

- Type inference
- Lambda functions and expressions
- Initializer lists / Template aliases
- Variadic templates
- Rvalue references and move constructors
- Others

# Type inference

auto creates a variable of the specific type of the initializer.

Before:

```
std::map<std::string, int> m;  
for(std::map<std::string, int>::iterator it = m.begin(); it != m.end(); ++it)  
    doit(*it);
```

After:

```
std::map<std::string, int> m;  
for(auto it = m.begin(); it != m.end(); ++it)  
    doit(*it);
```

# Lambda function and expressions

Before:

```
struct sum
{
    sum() : total(0) {}
    void operator()(int& n)
    {
        total += n;
    }
    int total;
};
```

```
sum s;
int arr[] = {1,2,3};
std::foreach(boost::begin(arr),
             boost::end(arr),
             s);
int total = s.total;
```

After:

```
int total;
int arr[] = {1,2,3};
std::foreach(boost::begin(arr),
             boost::end(arr),
             [&](int& n){ total += n; });
```

# Initializer lists / Template aliases

- Uniform initialization for all types:

```
int arr[3] = {1, 2, 3};  
vector<int> v{1, 2, 3};
```

- Template composition:

```
template <class T, unsigned Size>  
class array;
```

```
template <typename Size>  
using intarray = array<int, Size>;
```

```
intarray<10> arr; // same as array<int, 10>
```

# Variadic templates

## Templates with variable number of arguments

```
template <typename First>
First sum(First n)
{
    return n;
}
```

```
template <class First, class ...Rest>
First sum(First n, Rest... rest)
{
    return n + sum(rest...);
}
```

```
int ten = sum(2, 4, 3, 1);
```

More useful to library implementers.

# Rvalue references and move constructors

Before:

```
vector<int> get_numbers()
{
    vector<int> v;
    ....
    return v;
}
vector<int> numbers = get_numbers();
```

Two useless copies! (one to return from the function, one to initialize "numbers");

After:

```
void doit(string& lvalue);
void doit(string&& rvalue);

string s("hello");
doit(s);
doit(string("world"));
```

Zero copies \o/ !!!  
Move constructors make their members point to the rvalue's members and "zeroify" the rvalue (which thus won't be destroyed)

# Others

- Generalized constant expressions
- New string literals (unicode)
- User defined literals
- Multitasking memory model
- Library extensions (TR1/TR2... or "boost")

# Word of caution

*"C++: an octopus made by nailing extra legs onto a dog."*  
—Steve Taylor

- very long learning curve (never ending?)
- lots of corner cases
- can brain-damage you for life if you're not careful :)
- brutal syntax
- evolves slowly (try to put 30 übernerds in a room and ask them to agree on a feature).
- C++11 not completely here yet! (gcc/icc/clang "decent" and vc++ very limited)

# References & Questions?

<http://en.wikipedia.org/wiki/C%2B%2B>

<http://www.boost.org/>

<http://www2.research.att.com/~bs/>

[http://en.wikiquote.org/wiki/Bjarne\\_Stroustrup](http://en.wikiquote.org/wiki/Bjarne_Stroustrup)

[http://www.theregister.co.uk/2011/10/11/new\\_c\\_plus\\_plus\\_published/](http://www.theregister.co.uk/2011/10/11/new_c_plus_plus_published/)

<http://wiki.apache.org/stdcxx/C%2B%2B0xCompilerSupport>